Machine Learning Homework 5 Report

Gaussian Process and SVM

312554056 紀品榕

I. Gaussian Process

A. Code explanation

a. Libraries and modules imported

Libraries and modules imported into the program are shown below. The argparse library is imported for command-line options. The numpy module is imported for mathematical operations on arrays. The matplotlib.pyplot module is imported for visualization. The scipy.optimize module is used for minimization.

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

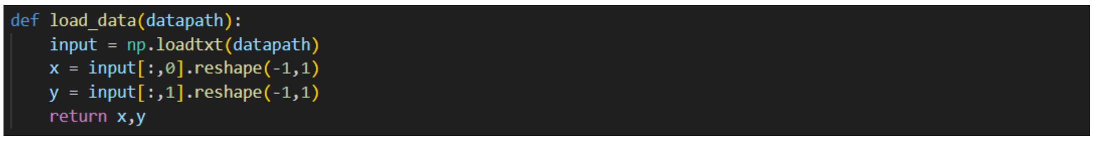
b. Overall workflow of the program

The data path of the input data and the noise for the function are set. The training points coordinate information x_train and y_train, obtained from the load_data function. The result visualization plotting space is also set. Then, the Gaussian process without parameters optimization is performed, and the result is visualized on the left side of the image. Afterward, the Gaussian process with parameters optimization by minimizing negative marginal log-likelihood is performed, and the result is visualized on the right side of the image. The final visualization result is then presented.

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Homework#05')
    parser.add_argument('--dataPath', type=str, default="input.data", help="path of the file")
    parser.add_argument('--b', type=float, default=5.0, help="noise for the function")
    setting = parser.parse_args()

    x_train, y_train = load_data(setting.dataPath)

    fig,ax = plt.subplots(1,2,figsize=(12,5))

    x_test,mean,var = perfrom_gaussian_process(x_train,y_train,setting.b,1.0,1.0)
    title = "alpha = 1.000, length scale = 1.000"
    draw_fig(ax[0],x_train,y_train,x_test,mean,var,title)


    init_par = [1.0,1.0]
    opt_par = minimize(calculate_neg_log_likelihood,init_par,args=(x_train,y_train,setting.b))

    x_test,mean,var = perfrom_gaussian_process(x_train,y_train,setting.b,opt_par.x[0],opt_par.x[1])
    title = "alpha =%.3f " %opt_par.x[0] +", length scale = %.3f"%opt_par.x[1]
    draw_fig(ax[1],x_train,y_train,x_test,mean,var,title)

    fig.tight_layout()
    plt.show()
```

c. Data loading

The load_data function loads the data to the x and the y arrays and reshapes them before returning the x and the y arrays. In this case, the x and the y arrays are in the shape of 34*1.



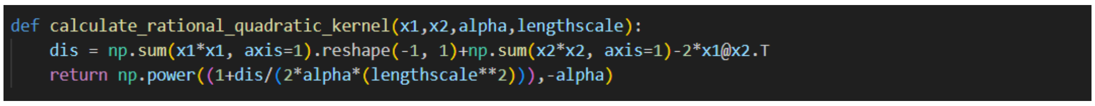
```python
def load_data(datapath):
    input = np.loadtxt(datapath)
    x = input[:,0].reshape(-1,1)
    y = input[:,1].reshape(-1,1)
    return x,y
```

d. Kernel calculating

The distance between the x1 and x2 vectors is calculated. Then, the rational quadratic kernel with variance 1 $\mathrm{k}(\mathrm{x}_1, \mathrm{x}_2)$ is calculated by the following formula:

$$\mathrm{k}(\mathrm{x}_1, \mathrm{x}_2) = \left(1 + \frac{\|x_1 - x_2\|^2}{2\alpha l^2}\right)^{-\alpha}$$

Where α represents the scale mixture parameter, and l represents the length scale of the kernel.



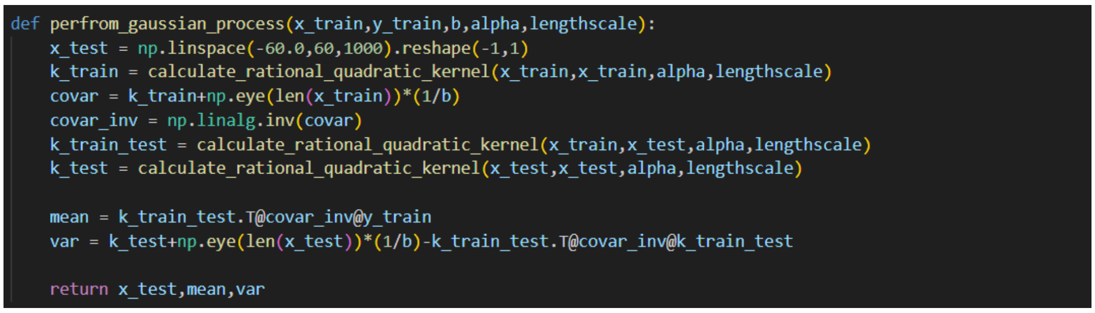
```python
def calculate_rational_quadratic_kernel(x1,x2,alpha,lengthscale):
    dis = np.sum(x1*x1, axis=1).reshape(-1, 1)+np.sum(x2*x2, axis=1)-2*x1@x2.T
    return np.power((1+dis/(2*alpha*(lengthscale**2))),-alpha)
```

e. Gaussian process

The Gaussian process is performed based on the solution as the following calculations:

$$\mu(\mathrm{x}^*) = k(x, x^*)^{\mathrm{T}} C^{-1} y$$
$$\sigma^2(x^*) = k^* - k(x, x^*)^{\mathrm{T}} C^{-1} k(x, x^*)$$
$$\mathrm{k}^* = k(x^*, x^*) + \beta^{-1}$$
$$\mathrm{C} = k(x, x) + \beta^{-1}$$

Where x represents the training data, x^* represents the testing data, C represents the covariance matrix of the training data, k^* represents the kernel of the testing data, and $k(x, x^*)$ represents the kernel between the training data and the testing data, and $\mu(\mathrm{x}^*)$ and $\sigma^2(x^*)$ are the mean and the variance of the predicted distribution respectively.



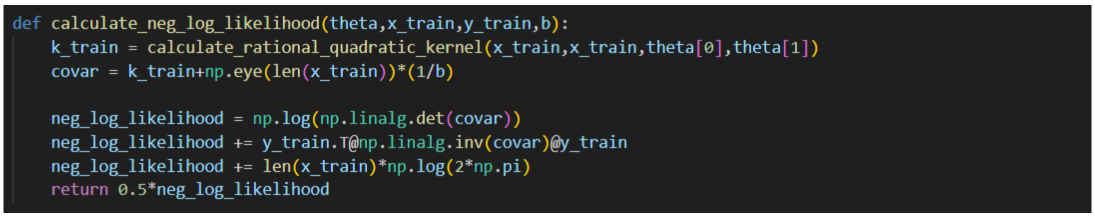
```python
def perfrom_gaussian_process(x_train,y_train,b,alpha,lengthscale):
    x_test = np.linspace(-60.0,60,1000).reshape(-1,1)
    k_train = calculate_rational_quadratic_kernel(x_train,x_train,alpha,lengthscale)
    covar = k_train+np.eye(len(x_train))*(1/b)
    covar_inv = np.linalg.inv(covar)
    k_train_test = calculate_rational_quadratic_kernel(x_train,x_test,alpha,lengthscale)
    k_test = calculate_rational_quadratic_kernel(x_test,x_test,alpha,lengthscale)

    mean = k_train_test.T@covar_inv@y_train
    var = k_test+np.eye(len(x_test))*(1/b)-k_train_test.T@covar_inv@k_train_test

    return x_test,mean,var
```

f. Negative marginal log-likelihood calculation

Minimizing negative marginal log-likelihood is performed for kernel parameters optimization. The negative marginal log-likelihood is calculated as below:

$$-\ln p(y|\theta) = \frac{1}{2}\ln|C_\theta| + \frac{1}{2}y^{\mathrm{T}}C_\theta^{-1}y + \frac{N}{2}\ln(2\pi)$$

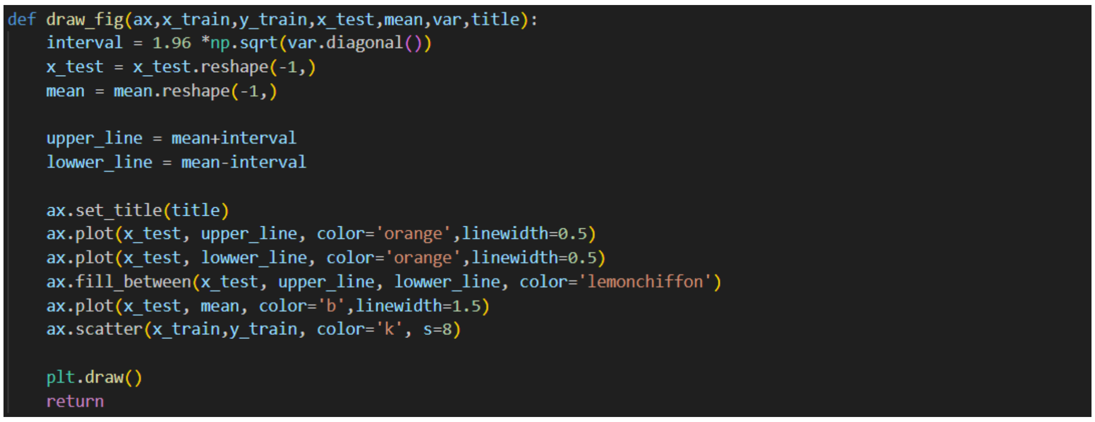
Where C_θ represents the covariance matrix of the training data, y represents the training label, and N represents the number of the training data.



```python
def calculate_neg_log_likelihood(theta,x_train,y_train,b):
    k_train = calculate_rational_quadratic_kernel(x_train,x_train,theta[0],theta[1])
    covar = k_train+np.eye(len(x_train))*(1/b)

    neg_log_likelihood = np.log(np.linalg.det(covar))
    neg_log_likelihood += y_train.T@np.linalg.inv(covar)@y_train
    neg_log_likelihood += len(x_train)*np.log(2*np.pi)
    return 0.5*neg_log_likelihood
```

g. Visualization

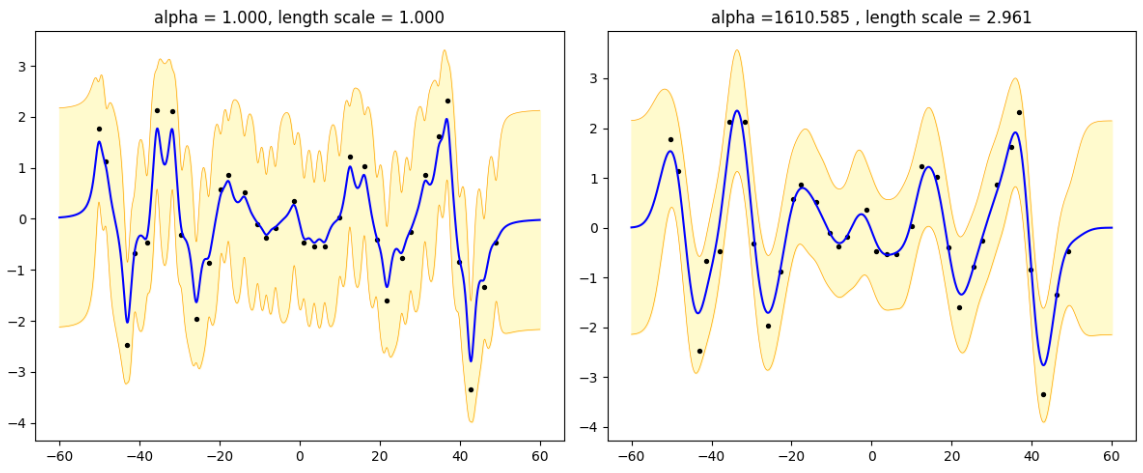
A 95% confidence interval means the interval is 1.96 times the variance. The 95% confidence interval is calculated and marked in lemon chiffon, and the testing and training data are plotted in blue and black, respectively.



```python
def draw_fig(ax,x_train,y_train,x_test,mean,var,title):
    interval = 1.96 *np.sqrt(var.diagonal())
    x_test = x_test.reshape(-1,)
    mean = mean.reshape(-1,)

    upper_line = mean+interval
    lowwer_line = mean-interval

    ax.set_title(title)
    ax.plot(x_test, upper_line, color='orange',linewidth=0.5)
    ax.plot(x_test, lowwer_line, color='orange',linewidth=0.5)
    ax.fill_between(x_test, upper_line, lowwer_line, color='lemonchiffon')
    ax.plot(x_test, mean, color='b',linewidth=1.5)
    ax.scatter(x_train,y_train, color='k', s=8)

    plt.draw()
    return
```

B. Results

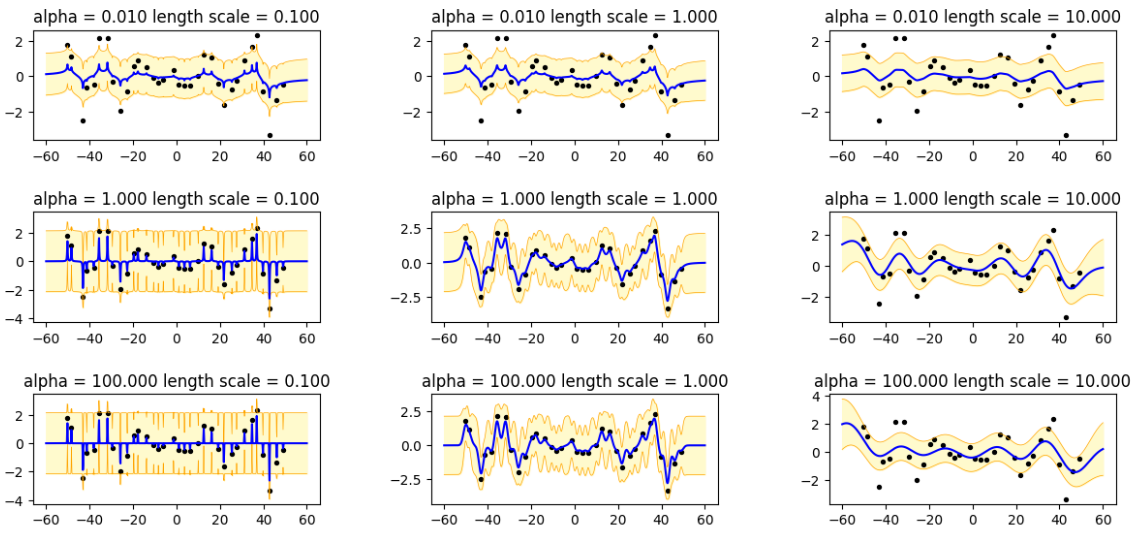
The left part of the figure shows the result of the Gaussian process with set kernel parameters by alpha and length scale parameters. The right part of the figure shows the result of the Gaussian process with the optimization of the kernel parameters, and the optimization results of alpha and length scale are shown as the title of the figure.



C. Observations and Discussion

a. After optimization, the fitting curve and the 95% confidence interval are smoother and narrower than the initial guess, thus showing generality to the testing data, better fitting ability, and more confidence in the prediction.

b. For the Scale mixture parameter α and the length scale parameter of the kernel l, an increase of α shows the larger length scale variation, and an increase of l leads to a less wiggly function.

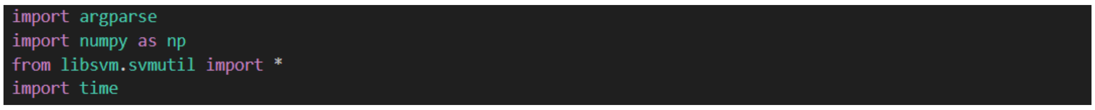


II. SVM on MNIST dataset

A. Code explanation

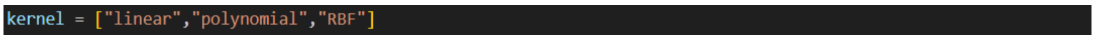
a. Libraries and modules imported, and the global variable

The argparse library is imported for command-line options. The numpy module is imported for mathematical operations on arrays. The libsvm module is imported for support vector machines. The time module is imported for time counting.



```python
import argparse
import numpy as np
from libsvm.svmutil import *
import time
```

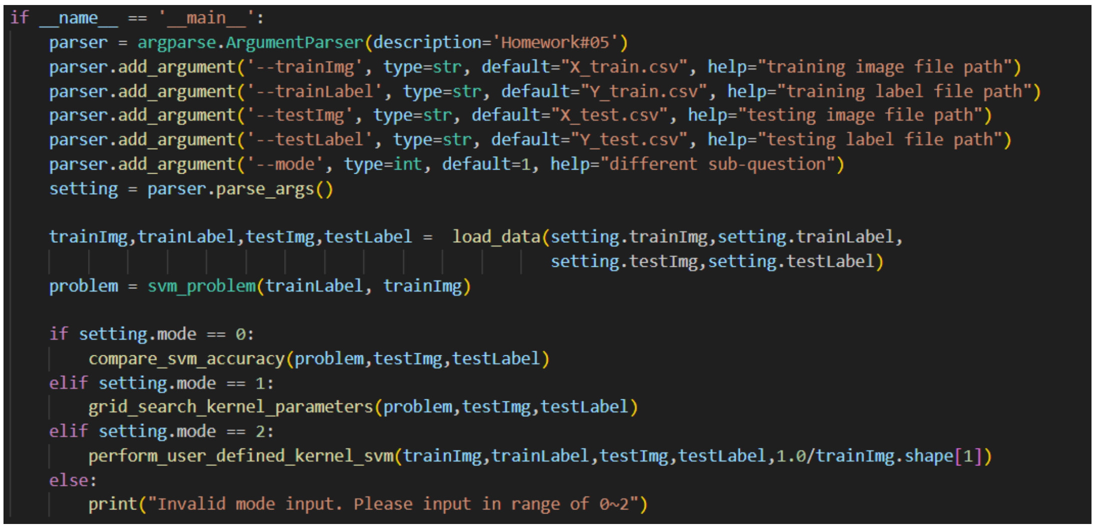
The global list variable kernel is set to map the kernel type with its option in the libsvm function by the list index.



```python
kernel = ["linear","polynomial","RBF"]
```

b. Overall workflow of the program

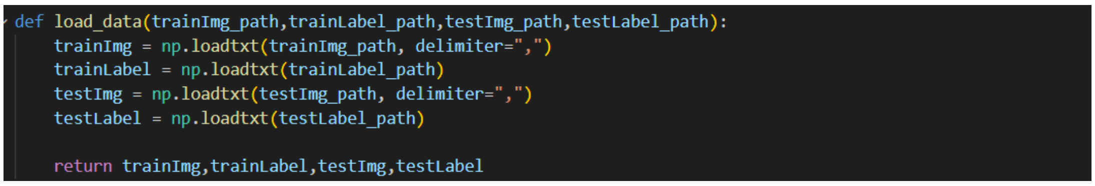
The data paths of the training images, training labels, testing images, and testing labels are set. The training and the testing data are loaded and stored as trainImg, trainLabel, testImg, and testLabel. The svm_problem instance of training data is then constructed by svm_problem. The mode option represents three questions, where mode 0 compares performance between different kernel functions, mode 1 performs grid search for different kernels in soft-margin SVM, and mode 2 uses a defined kernel to perform SVM.



```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Homework#05')
    parser.add_argument('--trainImg', type=str, default="X_train.csv", help="training image file path")
    parser.add_argument('--trainLabel', type=str, default="Y_train.csv", help="training label file path")
    parser.add_argument('--testImg', type=str, default="X_test.csv", help="testing image file path")
    parser.add_argument('--testLabel', type=str, default="Y_test.csv", help="testing label file path")
    parser.add_argument('--mode', type=int, default=1, help="different sub-question")
    setting = parser.parse_args()

    trainImg,trainLabel,testImg,testLabel =  load_data(setting.trainImg,setting.trainLabel,
                                                        setting.testImg,setting.testLabel)
    problem = svm_problem(trainLabel, trainImg)

    if setting.mode == 0:
        compare_svm_accuracy(problem,testImg,testLabel)
    elif setting.mode == 1:
        grid_search_kernel_parameters(problem,testImg,testLabel)
    elif setting.mode == 2:
        perform_user_defined_kernel_svm(trainImg,trainLabel,testImg,testLabel,1.0/trainImg.shape[1])
    else:
        print("Invalid mode input. Please input in range of 0~2")
```

c. Data loading

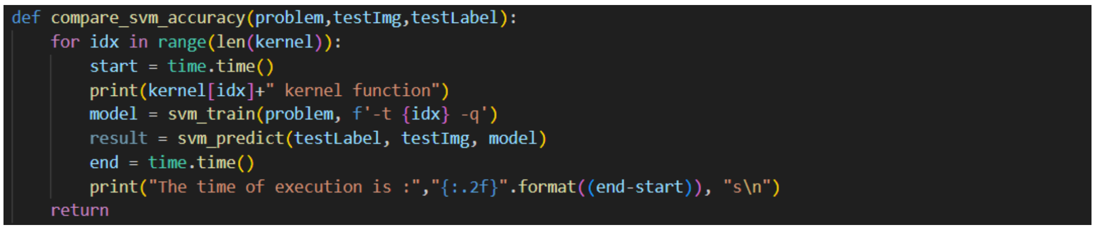
The load_data function loads the training data and label to the trainImg and the trainLabel arrays and loads the testing data and label to the testImg and the testLabel array. Then, the arrays are returned.



```python
def load_data(trainImg_path,trainLabel_path,testImg_path,testLabel_path):
    trainImg = np.loadtxt(trainImg_path, delimiter=",")
    trainLabel = np.loadtxt(trainLabel_path)
    testImg = np.loadtxt(testImg_path, delimiter=",")
    testLabel = np.loadtxt(testLabel_path)

    return trainImg,trainLabel,testImg,testLabel
```

d. Comparison between different kernel functions

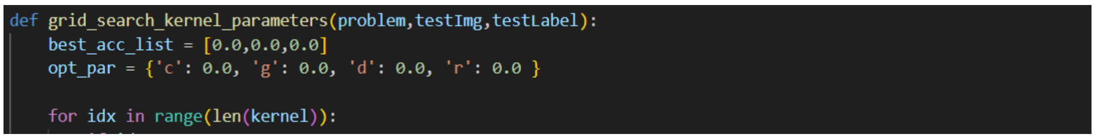
To compare the performance between the linear, polynomial, and RBF kernels, the start time of each calculation is stamped, and each model with a different kernel is trained by the svm_train function and used for prediction for testing data by the svm_predict function. After the prediction, the end time is stamped to calculate the execution time. Then, the prediction accuracy and execution time for each SVM with different kernel functions are shown.



```python
def compare_svm_accuracy(problem,testImg,testLabel):
    for idx in range(len(kernel)):
        start = time.time()
        print(kernel[idx]+" kernel function")
        model = svm_train(problem, f'-t {idx} -q')
        result = svm_predict(testLabel, testImg, model)
        end = time.time()
        print("The time of execution is :","{:.2f}".format((end-start)), "s\n")
    return
```

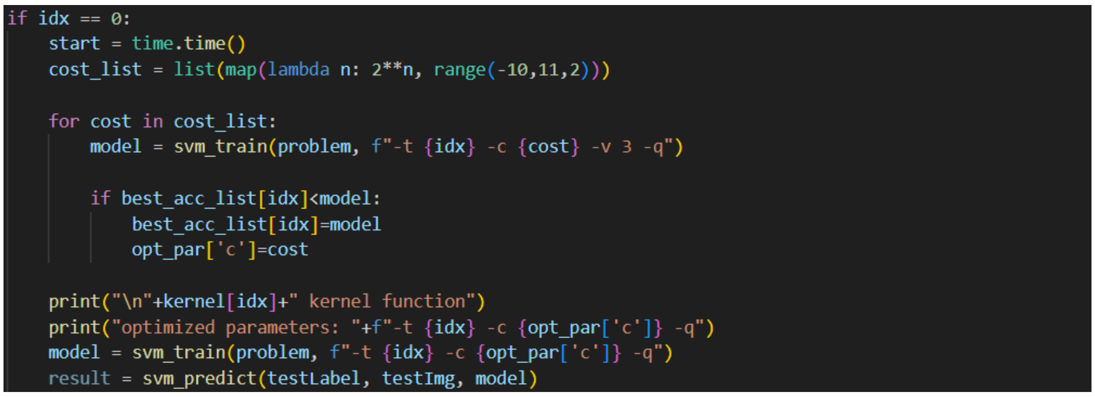
e. grid searching for finding parameters for C-SVM

The best_acc_list list records the best accuracy of C-SVM using the linear, polynomial, and RBF kernels. The opt_par dictionary records optimal parameters for the current kernel, where c represents the cost parameter of C-SVM, g represents the gamma in the kernel function, d represents the degree in the kernel function, and r represents the coef0 in the kernel function. C-SVM using the linear, polynomial, and RBF kernels with grid search for the best parameters are performed, respectively. Since C-SVM is the default SVM type for the library, the set type of SVM has not been changed.



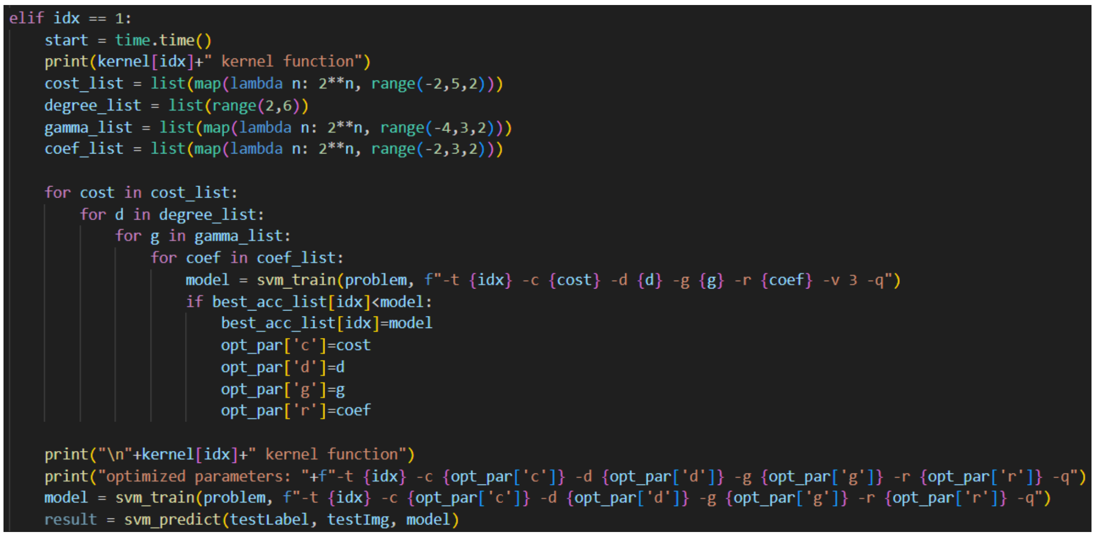
```python
def grid_search_kernel_parameters(problem,testImg,testLabel):
    best_acc_list = [0.0,0.0,0.0]
    opt_par = {'c': 0.0, 'g': 0.0, 'd': 0.0, 'r': 0.0 }

    for idx in range(len(kernel)):
```

Different cost parameters from a geometric sequence of 2^{-10} to 2^{10} with a common ratio of four are performed for the C-SVM with a linear kernel function. The -v parameter is specified for the number of cross-validation, and thus, the return of the svm_train function will be a scalar representing the training accuracy. The -q parameter is added to skip the printing of the training process. The parameter that results in the best training accuracy is then used in svm_train to obtain an instance of svm_model, and the model is used in the prediction.



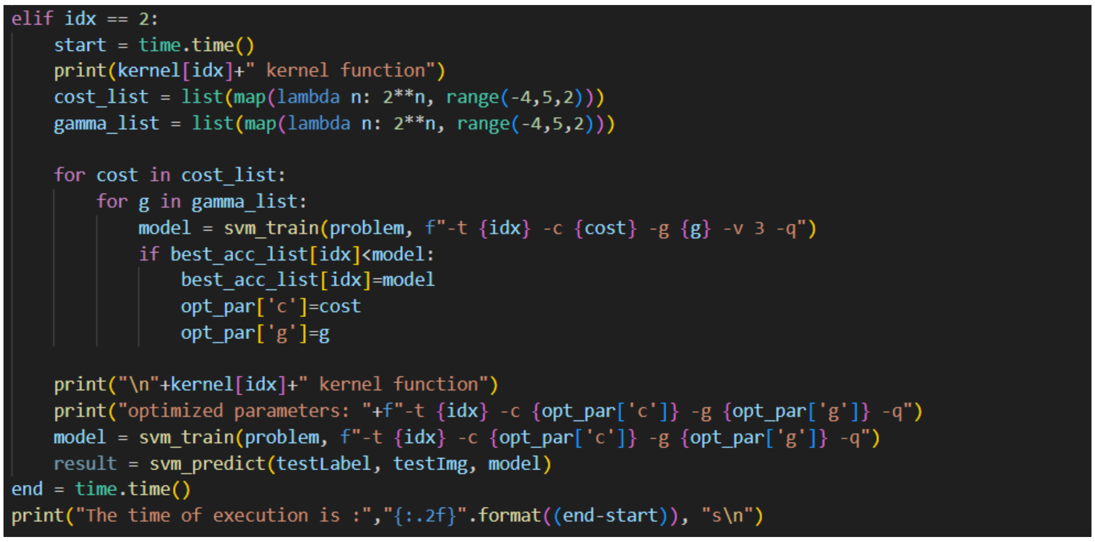
```python
if idx == 0:
    start = time.time()
    cost_list = list(map(lambda n: 2**n, range(-10,11,2)))

    for cost in cost_list:
        model = svm_train(problem, f"-t {idx} -c {cost} -v 3 -q")

        if best_acc_list[idx]<model:
            best_acc_list[idx]=model
            opt_par['c']=cost

    print("\n"+kernel[idx]+" kernel function")
    print("optimized parameters: "+f"-t {idx} -c {opt_par['c']} -q")
    model = svm_train(problem, f"-t {idx} -c {opt_par['c']} -q")
    result = svm_predict(testLabel, testImg, model)
```

For the polynomial kernel function, different cost parameters from a geometric sequence of 2^{-10} to 2^{10} with a common ratio of four, different degree parameters from 2 to 6, different gamma parameters from a geometric sequence of 2^{-4} to 2^{2} with a common ratio of four, and different coef0 parameters from a geometric sequence of 2^{-2} to 2^{2} with a common ratio of four are tested for the best training accuracy. The parameter pair with the best training accuracy is recorded in the opt_par dictionary, and the best training accuracy is recorded in best_acc_list. Then, the optimized parameters are used in svm_training to get the training model, and then the model is used for prediction.



```python
elif idx == 1:
    start = time.time()
    print(kernel[idx]+" kernel function")
    cost_list = list(map(lambda n: 2**n, range(-2,5,2)))
    degree_list = list(range(2,6))
    gamma_list = list(map(lambda n: 2**n, range(-4,3,2)))
    coef_list = list(map(lambda n: 2**n, range(-2,3,2)))

    for cost in cost_list:
        for d in degree_list:
            for g in gamma_list:
                for coef in coef_list:
                    model = svm_train(problem, f"-t {idx} -c {cost} -d {d} -g {g} -r {coef} -v 3 -q")
                    if best_acc_list[idx]<model:
                        best_acc_list[idx]=model
                        opt_par['c']=cost
                        opt_par['d']=d
                        opt_par['g']=g
                        opt_par['r']=coef

    print("\n"+kernel[idx]+" kernel function")
    print("optimized parameters: "+f"-t {idx} -c {opt_par['c']} -d {opt_par['d']} -g {opt_par['g']} -r {opt_par['r']} -q")
    model = svm_train(problem, f"-t {idx} -c {opt_par['c']} -d {opt_par['d']} -g {opt_par['g']} -r {opt_par['r']} -q")
    result = svm_predict(testLabel, testImg, model)
```

For the RBF function, different cost parameters from a geometric sequence of 2^{-10} to 2^{10} with a common ratio of four and different gamma parameters from a geometric sequence of 2^{-4} to 2^{2} with a common ratio of four. The parameter pair with the best training accuracy is recorded in the opt_par dictionary, and the best training accuracy is recorded in best_acc_list. Then, the optimized parameters are used in svm_training to get the training model, and then the model is used for prediction.

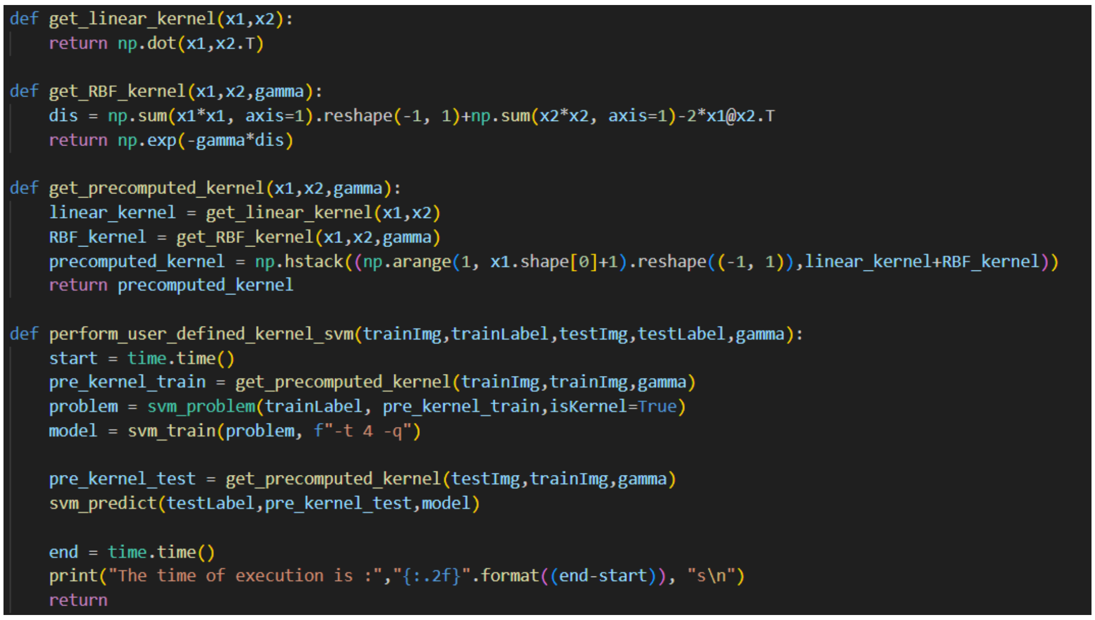


```python
elif idx == 2:
    start = time.time()
    print(kernel[idx]+" kernel function")
    cost_list = list(map(lambda n: 2**n, range(-4,5,2)))
    gamma_list = list(map(lambda n: 2**n, range(-4,5,2)))

    for cost in cost_list:
        for g in gamma_list:
            model = svm_train(problem, f"-t {idx} -c {cost} -g {g} -v 3 -q")
            if best_acc_list[idx]<model:
                best_acc_list[idx]=model
                opt_par['c']=cost
                opt_par['g']=g

    print("\n"+kernel[idx]+" kernel function")
    print("optimized parameters: "+f"-t {idx} -c {opt_par['c']} -g {opt_par['g']} -q")
    model = svm_train(problem, f"-t {idx} -c {opt_par['c']} -g {opt_par['g']} -q")
    result = svm_predict(testLabel, testImg, model)
end = time.time()
print("The time of execution is :","{:.2f}".format((end-start)), "s\n")
```

f. Using a user-defined kernel in SVM

The get_linear_kernel function is used to get the linear kernel of two input matrixes, x_1 and x_2. The get_RBF_kernel function is used to get the RBF kernel of two input matrixes, x_1 and x_2, with the input gamma parameter. The get_precomputed_kernel function is used to construct a kernel that adds the linear and the RBF kernel together and adds the index in front of the kernel to satisfy the format of the precomputed training or testing kernel for the library.

The perform_user_defined_kerenl_svm function uses training data and the input gamma parameter to compute the training precomputed kernel. The precomputed kernel is then used in svm_problem to get the training model. The isKernel parameter must be set as true since using a precomputed kernel. The precomputing testing kernel is calculated and used with the prediction model.

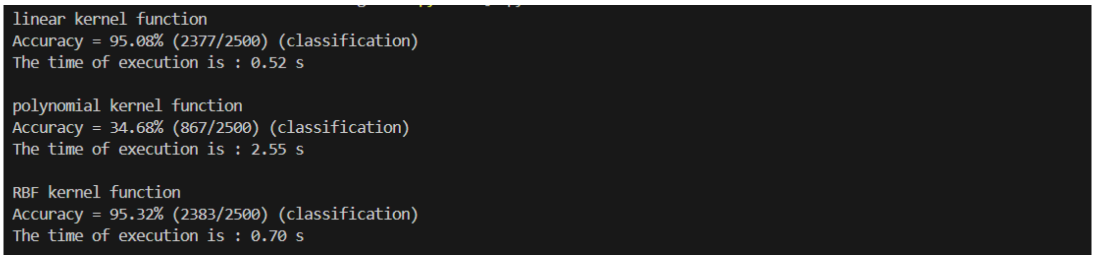


```python
def get_linear_kernel(x1,x2):
    return np.dot(x1,x2.T)

def get_RBF_kernel(x1,x2,gamma):
    dis = np.sum(x1*x1, axis=1).reshape(-1, 1)+np.sum(x2*x2, axis=1)-2*x1@x2.T
    return np.exp(-gamma*dis)

def get_precomputed_kernel(x1,x2,gamma):
    linear_kernel = get_linear_kernel(x1,x2)
    RBF_kernel = get_RBF_kernel(x1,x2,gamma)
    precomputed_kernel = np.hstack((np.arange(1, x1.shape[0]+1).reshape((-1, 1)),linear_kernel+RBF_kernel))
    return precomputed_kernel

def perform_user_defined_kernel_svm(trainImg,trainLabel,testImg,testLabel,gamma):
    start = time.time()
    pre_kernel_train = get_precomputed_kernel(trainImg,trainImg,gamma)
    problem = svm_problem(trainLabel, pre_kernel_train,isKernel=True)
    model = svm_train(problem, f"-t 4 -q")

    pre_kernel_test = get_precomputed_kernel(testImg,trainImg,gamma)
    svm_predict(testLabel,pre_kernel_test,model)

    end = time.time()
    print("The time of execution is :","{:.2f}".format((end-start)), "s\n")
    return
```

B. Results

a. Comparison between different kernel functions

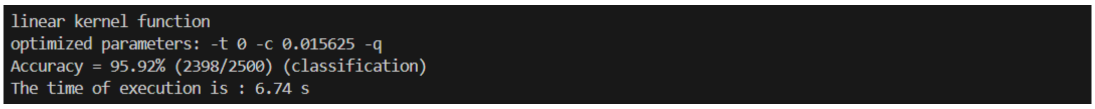
The testing accuracy and execution time of linear, polynomial, and RBF kernel functions with default parameter settings are shown below.



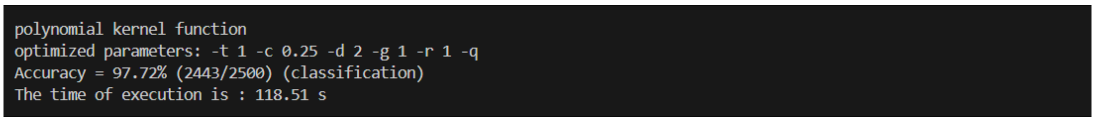
```
linear kernel function
Accuracy = 95.08% (2377/2500) (classification)
The time of execution is : 0.52 s

polynomial kernel function
Accuracy = 34.68% (867/2500) (classification)
The time of execution is : 2.55 s

RBF kernel function
Accuracy = 95.32% (2383/2500) (classification)
The time of execution is : 0.70 s
```

b. grid searching for finding parameters for C-SVM with different kernels

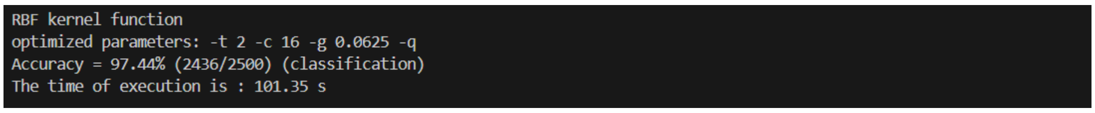
The optimized parameters found by grid search, the testing accuracy, and the execution time of linear, polynomial, and RBF kernel functions are shown below.



```
linear kernel function
optimized parameters: -t 0 -c 0.015625 -q
Accuracy = 95.92% (2398/2500) (classification)
The time of execution is : 6.74 s
```



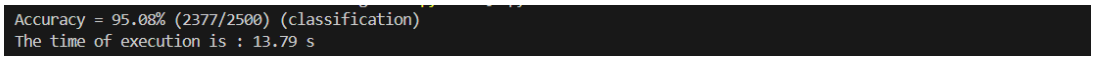
```
polynomial kernel function
optimized parameters: -t 1 -c 0.25 -d 2 -g 1 -r 1 -q
Accuracy = 97.72% (2443/2500) (classification)
The time of execution is : 118.51 s
```



```
RBF kernel function
optimized parameters: -t 2 -c 16 -g 0.0625 -q
Accuracy = 97.44% (2436/2500) (classification)
The time of execution is : 101.35 s
```

c. Using a user-defined kernel in SVM

The testing accuracy and execution time of user-defined kernel combining linear and RBF kernel functions are shown below. The gamma parameter for the RBF kernel is set the same as the parameter in the library; $\frac{1}{\text{number of the features}}$.



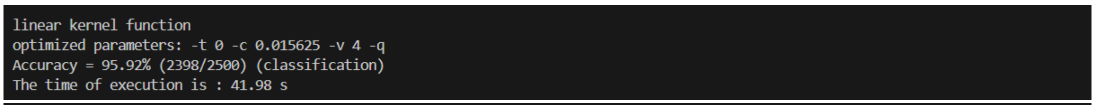
```
Accuracy = 95.08% (2377/2500) (classification)
The time of execution is : 13.79 s
```

C. Observations and Discussion

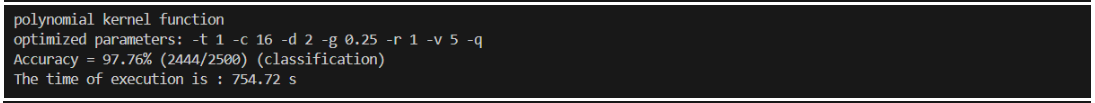
a. Linear and RBF kernel functions with default parameters perform well for testing accuracy, while the polynomial kernel function does not perform well in classification. After the grid search, three SVMs with different kernel functions performed better than before the optimization.

b. The polynomial kernel function is more time-consuming than the others. Regarding grid search, the number of hyper-parameters for the polynomial kernel function is more than the others, leading to a longer execution time.

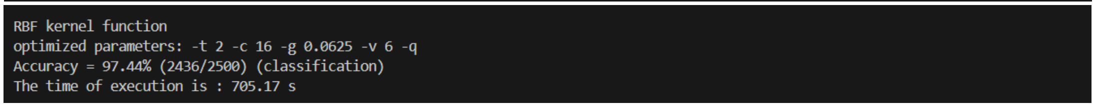
c. The number of cross-validation seems not to impact the parameter choice greatly. The optimized parameters of linear and RBF kernel functions are the same. Furthermore, the training model of getting the best validation result may not get the best testing data accuracy.



```
linear kernel function
optimized parameters: -t 0 -c 0.015625 -v 4 -q
Accuracy = 95.92% (2398/2500) (classification)
The time of execution is : 41.98 s
```



```
polynomial kernel function
optimized parameters: -t 1 -c 16 -d 2 -g 0.25 -r 1 -v 5 -q
Accuracy = 97.76% (2444/2500) (classification)
The time of execution is : 754.72 s
```



```
RBF kernel function
optimized parameters: -t 2 -c 16 -g 0.0625 -v 6 -q
Accuracy = 97.44% (2436/2500) (classification)
The time of execution is : 705.17 s
```